Bitmap indices are a special type of index designed for efficient querying on multiple keys.

Records in a relation are assumed to be numbered sequentially from, say, 0.
- Given a number \( n \) it must be easy to retrieve record \( n \).

Applicable on attributes that take on a relatively small number of distinct values:
- E.g., gender, country, state, ...
- E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity).

A bitmap is simply an array of bits.
In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute

- A bitmap has as many bits as records
- In a bitmap for value \( v \), the bit for a record is 1 if the record has the value \( v \) for the attribute, and is 0 otherwise

<table>
<thead>
<tr>
<th>record number</th>
<th>name</th>
<th>gender</th>
<th>address</th>
<th>income-level</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>John</td>
<td>m</td>
<td>Perryridge</td>
<td>L1</td>
</tr>
<tr>
<td>1</td>
<td>Diana</td>
<td>f</td>
<td>Brooklyn</td>
<td>L2</td>
</tr>
<tr>
<td>2</td>
<td>Mary</td>
<td>f</td>
<td>Jonestown</td>
<td>L1</td>
</tr>
<tr>
<td>3</td>
<td>Peter</td>
<td>m</td>
<td>Brooklyn</td>
<td>L4</td>
</tr>
<tr>
<td>4</td>
<td>Kathy</td>
<td>f</td>
<td>Perryridge</td>
<td>L3</td>
</tr>
</tbody>
</table>

Bitmaps for gender

- \( m \)
  - \( 10010 \)

- \( f \)
  - \( 01101 \)

Bitmaps for income-level

- \( L1 \)
  - \( 10100 \)

- \( L2 \)
  - \( 01000 \)

- \( L3 \)
  - \( 00001 \)

- \( L4 \)
  - \( 00010 \)

- \( L5 \)
  - \( 00000 \)
• Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries

• Queries are answered using bitmap operations
  - Intersection (AND)
  - Union (OR)
  - Complementation (NOT)

• Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g.: Males with income level L1: 10010 AND 10100 = 10000
    • Can then retrieve required tuples
    • Counting number of matching tuples is even faster (especially useful for aggregation queries)
Bitmap indices generally very small compared with relation size

**Deletion needs to be handled properly**
- We must make use of an *existence bitmap* to note if there is a valid record at a record location
- Needed for *complementation*
  - $\text{NOT}(A=v): \ (\text{NOT bitmap-A-v}) \ \text{AND} \ \text{ExistenceBitmap}$

**Should keep bitmaps for all values, even null values**
- To correctly handle SQL null semantics for
  - $\text{NOT}(A=v): \ (\text{NOT bitmap-A-v}) \ \text{AND} \ \text{ExistenceBitmap} \ \text{AND} \ \text{NOT(bitmap-A-Null)}$
SELECT stu_id FROM scores
WHERE maths in [80, 90] AND chem in [80, 90]

• If we have an index on maths:
  - First retrieve all tuples satisfying maths in [80, 90]
  - Check each such tuple about chemistry
  - Problem: Many tuples satisfying maths in [80, 90] do not satisfy chem in [80, 90] but need to be checked anyway

• The case of an index for chem is similar

• If we have *two* indexes (one on maths and another on chem), we can do pointer intersection before retrieving the actual records
• Grid files are used to speed up the processing of general multiple search-key queries involving one or more comparison operators

• The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.

• Multiple cells of grid array can point to the same bucket

• To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer
Example Grid File for *Account*

- **Townsend**
- **Perryridge**
- **Mianus**
- **Central**

Linear scale for *branch-name*

Grid Array:
- 4
- 3
- 2
- 1
- 0

Linear scale for *balance*

<table>
<thead>
<tr>
<th>1K</th>
<th>2K</th>
<th>5K</th>
<th>10K</th>
<th>50K</th>
<th>100K</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Buckets

kwtleung@cse.ust.hk

CSIT5300
• A grid file on two attributes $A$ and $B$ can handle queries of all following forms with reasonable efficiency
  - $(a_1 \leq A \leq a_2)$
  - $(b_1 \leq B \leq b_2)$
  - $(a_1 \leq A \leq a_2 \land b_1 \leq B \leq b_2)$

• E.g., to answer $(a_1 \leq A \leq a_2 \land b_1 \leq B \leq b_2)$, use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells
During insertion, if a bucket becomes full, a new bucket can be created
- Idea similar to extendible hashing, but on multiple dimensions
- If only one cell points to it, either an overflow bucket must be created or the grid size must be increased

Linear scales must be chosen to uniformly distribute records across cells
- Otherwise there will be too many overflow buckets

Periodic re-organization to increase grid size will help
- But reorganization can be very expensive

Space overhead of grid array can be high
Basic Steps in Query Processing

- **Parsing and translation**
  - Query is checked for syntax, relations are verified and query is translated into relational algebra

- **Evaluation**
  - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query
• A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\pi_{balance}(Account))$ is equivalent to $\pi_{balance}(\sigma_{balance<2500}(Account))$

• Each relational algebra operation can be evaluated using one of several different algorithms, e.g., the
  - can use an index on $balance$ to find accounts with balance $<2500$, or
  - can perform complete relation scan and discard accounts with balance $\geq 2500$

• An annotated expression specifying detailed evaluation strategy is called an evaluation-plan
Query Optimization: Amongst all equivalent evaluation plans choose the one with the (expected) lowest cost
  - Cost is estimated using statistical information from the database, e.g. number of tuples in each relation, size of tuples, etc.

For simplicity we just use number of page transfers from disk as the cost measure
  - We ignore the difference in cost between sequential and random I/O for simplicity. We also ignore CPU costs for simplicity

Costs depend on the size of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and is hard to determine
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account
We do not include cost of writing the final output to disk
Selection Operation

- **File scan**: search algorithms that locate and retrieve records that fulfill a selection condition

- Algorithm **A1** (linear search): Scan each page and test all records to see whether they satisfy the selection condition
  - Cost estimate (number of disk pages scanned) = $b_r$
    - $b_r$ denotes number of pages containing records from relation $r$
  - If selection is equality on a key attribute, average cost = $(b_r/2)$
    - stop on finding (the unique) record
  - Linear search can be applied regardless of
    - selection condition, or
    - ordering of records in the file, or
    - availability of indices
• **A2 (binary search):** Applicable if selection is an equality comparison on the attribute on which the file is ordered
  - Assume that the pages of a relation are stored contiguously
  - Cost estimate (number of disk pages to be scanned):
    - $\left\lceil \log_2(b_r) \right\rceil$: cost of locating the first tuple by a binary search on the pages
    - *plus* number of pages containing records that satisfy the selection condition
Selection Using Indices

- **A3** (primary index on candidate key, equality): Retrieve a single record that satisfies the corresponding equality condition
  - *Cost* = *HT*<sub>i</sub> + 1
    - *HT*<sub>i</sub> is the height of the tree index
    - if we use a hash index, *HT*<sub>i</sub> = 1 or *HT*<sub>i</sub> = 1.2 if we assume that there exist overflow buckets

- **A4** (primary index on nonkey, equality): Retrieve multiple records
  - Records will be on consecutive pages
  - *Cost* = *HT*<sub>i</sub> + number of pages containing retrieved records

- **A5** (equality on search-key of secondary index)
  - Retrieve a single record if the search-key is a candidate key
    - *Cost* = *HT*<sub>i</sub> + 1
  - Retrieve multiple records if search-key is not a candidate key
    - *Cost* = *HT*<sub>i</sub> + number of records retrieved
    - Can be very expensive!
      - Each record may be on a different page
      - One page access for each retrieved record
Can implement selections of the form $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$ by using
- a linear file scan or binary search,
- or by using indices in the following ways:

**A6 (primary index, comparison):** (relation is sorted on A)
- For $\sigma_{A \geq V}(r)$ use index to find first tuple $\geq v$ and scan relation sequentially from there
- For $\sigma_{A \leq V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index

**A7 (secondary index, comparison):**
- For $\sigma_{A \geq V}(r)$ use index to find first index entry $\geq v$ and scan index sequentially from there, to find pointers to records
- For $\sigma_{A \leq V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
- In either case, retrieve records that are pointed to
  - requires an I/O for each record
  - linear file scan may be cheaper if many records are to be fetched!
Conjunction: $\sigma_{\theta_1 \land \theta_2 \land \ldots \land \theta_n}(r)$

**A8** (conjunctive selection using one index):
- Select a combination of $\theta_i$ and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta_i}(r)$
- Test other conditions on each tuple after fetching it into the memory buffer

**A9** (conjunctive selection using multiple-key index):
- Use appropriate composite (multiple-key) index if available

**A10** (conjunctive selection by intersection of identifiers):
- Requires indices with record pointers
- Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers
- Then fetch records from file
- If some conditions do not have appropriate indices, apply test in memory
Disjunction: $\sigma_{\theta_1} \lor \sigma_{\theta_2} \lor \ldots \lor \sigma_{\theta_n}(r)$.

- **A11** *(disjunctive selection by union of identifiers)*:
  - Applicable if *all* conditions have available indices
    - Otherwise use linear scan
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers
  - Then fetch records from file
**Example**: Merging sorted files with 3 pages of main memory buffer

<table>
<thead>
<tr>
<th>sorted file 1</th>
<th>sorted file 2</th>
<th>merged file</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,...)</td>
<td>(2,...)</td>
<td>(1,...)</td>
</tr>
<tr>
<td>(5,...)</td>
<td>(4,...)</td>
<td>(2,...)</td>
</tr>
<tr>
<td>(7,...)</td>
<td>(6,...)</td>
<td>(4,...)</td>
</tr>
<tr>
<td>(11,...)</td>
<td>(9,...)</td>
<td>(5,...)</td>
</tr>
<tr>
<td>(12,...)</td>
<td>(10,...)</td>
<td></td>
</tr>
<tr>
<td>(15,...)</td>
<td>(14,...)</td>
<td></td>
</tr>
<tr>
<td>(20,...)</td>
<td>(17,...)</td>
<td></td>
</tr>
<tr>
<td>(21,...)</td>
<td>(18,...)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(6,...)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(7,...)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(9,...)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(11,...)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
External Sorting (Disk-resident Files)

<table>
<thead>
<tr>
<th>sorted file 1</th>
<th>sorted file 2</th>
<th>merged file</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,...)</td>
<td>(2,...)</td>
<td>(1,...)</td>
</tr>
<tr>
<td>(5,...)</td>
<td>(4,...)</td>
<td>(2,...)</td>
</tr>
<tr>
<td>(7,...)</td>
<td>(6,...)</td>
<td>(4,...)</td>
</tr>
<tr>
<td>(11,...)</td>
<td>(9,...)</td>
<td>(5,...)</td>
</tr>
</tbody>
</table>

Bring the first page from each file

write page to disk

(6,...)
(7,...)
(9,...)
## External Sorting (Disk-resident Files)

<table>
<thead>
<tr>
<th>sorted file 1</th>
<th>sorted file 2</th>
<th>merged file</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,...)</td>
<td>(2,...)</td>
<td>(1,...)</td>
</tr>
<tr>
<td>(5,...)</td>
<td>(4,...)</td>
<td>(2,...)</td>
</tr>
<tr>
<td>(7,...)</td>
<td>(6,...)</td>
<td>(4,...)</td>
</tr>
<tr>
<td>(11,...)</td>
<td>(9,...)</td>
<td>(5,...)</td>
</tr>
</tbody>
</table>

Bring next page of file 2

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>write page to disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10,...)</td>
<td>(6,...)</td>
<td></td>
</tr>
<tr>
<td>(14,...)</td>
<td>(7,...)</td>
<td></td>
</tr>
<tr>
<td>(17,...)</td>
<td>(9,...)</td>
<td></td>
</tr>
<tr>
<td>(18,...)</td>
<td>(10,...)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(11,...)</td>
<td></td>
</tr>
</tbody>
</table>
### External Sorting (Disk-resident Files)

#### sorted file 1
- (1,...)  
- (5,...)  
- (7,...)  
- (11,...)  

#### sorted file 2
- (2,...)  
- (4,...)  
- (6,...)  
- (9,...)  

#### merged file
- (1,...)  
- (2,...)  
- (4,...)  
- (5,...)  
- (6,...)  
- (7,...)  
- (9,...)  
- (10,...)  
- (11,...)  
- (12,...)  

**Operation:**
- Write page to disk

**Next Steps:**
- Bring next page of file 1
- Bring next page of file 2

**Note:**
- The diagram illustrates the process of merging two sorted files into a single merged file, with pages being written to disk as needed.
Continuing the previous example:

- **Question:** We assumed that each file is already sorted. If the file is not sorted, how do we sort it (using only 3 buffer pages?)
  - **Answer:** Each file in the example is only 2 pages. Therefore, we can bring the entire file in memory, and sort it using any main-memory algorithm.

- **Question:** The previous example assumes two separate files. How do I apply this idea to sort a single file?
  - **Answer:** You can split the file in two parts and merge them as if they were separate files.

- **Question:** Can I do better if I have $M > 3$ main memory pages?
  - **Answer:** Yes, instead of 2 you can merge up to $M - 1$ files (because you need 1 page for writing the output).
Let $M$ denote memory size (in pages).

• **Create sorted runs.** Let $i$ be 0 initially. Repeat the following till the end of the relation:
  (a) Read $M$ pages of the relation into memory
  (b) Sort the in-memory pages
  (c) Write sorted data to run $R_i$; increment $i$
Let the final value of $i$ be $N$

• **Merge the runs (N-way merge).** We assume (for now) that $N < M$. Use $N$ pages of memory to buffer input runs, and 1 page to buffer the output. Read the first page of each run into its buffer page

```repeat
  Select the first record (in sorted order) among all buffer pages
  Write the record to the output buffer
  If the output buffer is full then flush it to disk
  Delete the record from its input buffer page.
  If an input buffer page becomes empty then read the next page (if any) of the run into the buffer
```

until all input buffer pages are empty
If $N \geq M$, several merge passes are required

- In each pass, contiguous groups of $M-1$ runs are merged
- A pass reduces the number of runs by a factor of $M-1$, and creates runs longer by the same factor
  - E.g., if $M=11$, and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
- Repeated passes are performed till all runs have been merged into one
- Total cost = $b_r(2\lceil \log_{M-1}(b_r/M) \rceil + 1)$
  - where $b_r$ is the file size in pages
  - the last pass does not write the result back to the disk
External Sort-Merge – Example

- 1 record per page
- 3 buffer pages